Name: Adem Atmaca Matriculation Nr.: 5502835	Name: Adrian Sauter Matriculation Nr.: 5458958

Q1	Q2	Q3	Q4	Sum

Artificial Intelligence: Assignment 8

0.1 Basis of our approach

The algorithm we decided to go with makes use of the alpha-beta-pruning approach. We started out by implementing the minimax-algorithm from the lecture and then changed the key parts to make the algorithm work in the same way as the alpha-beta-pruning algorithm from the lecture does. We iterated over all possible Moves (via gs.getValidMoves) and simulated each possible move (vie gs.makeMove (later reset via gs.undoMove) and then calling the corresponding alpha-beta-min or alpha-beta-max function accordingly.
We set the maxDepth to 4 after experimenting with different depths and learning that the maxDepth of 4 leads to finding good moves in the given time window. The maxDepth of 4 means that the agent will simulate his own move, then the opponents, then another one of his own and finally one more move by the opponent. There were two possible reasons that the interpolation between alpha-beta-min and alpha-beta-max would be broken:
First, it could be the case that the maxDepth was reached. If that was the case, we wanted to evaluate the gamestate that was reached after the moves that were simulated in the process.
In the other case, we have do differentiate between max and min: In the case of max (which represents our agent), we checked whether the list of validMoves is empty, in which case a scenario would have led to our agent losing or a stalemate. We then returned negativeInfinity so that our agent would not choose a move that leads to that scenario. In the case of min (the opponent), if there were no validMoves left, the opponent would certainly be in a checkmate or stalemate situation. In the first case, we would certainly want to make this move since it could lead to a promising outcome (i.e. winning).

0.2 Evaluation function

In the following section, we will explain how we developed our evaluation function: We first started out with a simple technique to evaluate the board by assigning each type of piece a certain value, where positive values indicated own pieces while negative values indicated the opponents pieces. Own pieces were valued a little bit higher (10%) then enemy pieces so that our agent wouldn't always choose to simply trade pieces. The values were inferred from the original chess piece values and multiplied by 100. The values can be seen in Table 1:

	Pawn	Knight	Bishop	Rook	King
Own pieces	110	330	330	550	11000
Enemy pieces	-100	-300	-300	-500	-1000

Table 1: Assigned piece values to own/ enemy pieces correspondingly

These piece values were then used to evaluate the board the following way: We iterated over each square and if it was not empty, added the value of our piece or subtracted the value of the opponents piece, respectively. By using this to evaluate the board in combination with the alpha-beta-pruning, our agent would try to protect his own pieces while attacking the pieces of the opponent. Since the king of the opponent had the lowest value, he was automatically considered the most valuable/ desirable piece (and the own king was regarded as the piece that has to be protected at all costs).

0.3 Further evaluation techniques

We added some further evaluation techniques to make our agents choices more intelligent which will be explained in the following paragraphs.

0.3.1 Piece square tables

We added piece-square tables to reward/punish good/bad positions of the pieces. Each piece (except for the rook) got his own table which assigned each square of the board a positive/negative value (or 0 if it was neutral). This value represented the value of the piece being placed on this certain square. We came up with the values by thinking about the moving abilities of the different pieces and judging where a certain piece is more useful for the game (i.e. danger to the opponent, protection of the own king, development

of pieces, etc). A negative value therefore corresponds to a bad position of this certain piece. These values were then added/subtracted from the overall utiliy_value which was calculated in the evaluation-function. We also used these tables to make the pieces of the opponent more/ less valuable according to their position. By doing so, our agent should attack more dangerously located enemy pieces more than less dangerous pieces. However, to make this influence not too high, we only took half of the values of the piece-square tables to add/subtract from the utility_value.
We differentiated between most of the game in contrast to the endgame. Take for example the own king, which should stick to the corners for most of the game to not be exposed to enemy attacks, but move to the middle towards the endgame so that he can't be easily trapped in a corner. The endgame was reached after a specified lower boundary of pieces left on the board was reached. The endgame was determined by a lower boundary (i.e. 24000) of accumulated piece values left of the board.

0.3.2 King safety

If the own king was positioned in one of the corners of "own side" of the board for example by castling (which was made attractive by the piece-square tables), we updated the piece-square tables of our own pawns so that they would stay in front of the king in order to build a defensive line in front of him, protection him from forward attacks. Moving upwards by one square wasn't punished very hard since that wouldn't destroy the entire defense, but moving any further was made undesirable by low positive values in contrast to the high positive values for the squares in front of the king.

0.3.3 Piece values considering the number of pawns

In the case of few pawns (less than 4 total pawns left on the board (own and enemy pawns)), rooks and bishops should be considered more valuable since they have more room to move around. We included this by updating the piece values accordingly. Also, if there are few pawns left, the pawns should be considered more valuable since loosing all pawns results in not being able to make use of pawn promotion which is an important aspect of many endgames.
In the case of many pawns (4 or more total pawns left on the board), knights are very valuable since they can easily escape "tight" situations by jumping over pieces. Therefore, we updated the weights of knights to a higher value when there were many pawns left.

0.3.4 Killer instinct

In case our agent could set the opponent to checkmate with the first hypothetical move (i.e. in depth = 0), we encouraged our agent to definitely choose that move by saving it to the class variable "checkmateMove". This move was therefore set (or not set) in the first call of the alpha-beta-max-function. In the findBestMove-function, we then always asked whether the "checkmateMove" existed or not. If he existed, our agent would update the queue with that move.

0.3.5 Try to force threefold repetition

In case our agent only has his king left (and therefore can't win), we implemented a way for him to try to force a draw by threefold repetition. We did this by checking whether we can reach the same board state we had before our last move by moving our king in our current turn. If this was the case, then he should certainly do that move. Otherwise, he should use the regular alpha-beta-pruning way to get his move.

0.3.6 Further additions

We added some further things that don't fall under one of the categories above but are explained here:

Our agent tries to avoid threefold repetition (unless it's the best possible outcome, see 0.3.5) by choosing the second best move in case the best move leads to the same board state as before. We are aware of the fact that this doesn't maximize the performance of our agent (since he then chooses the second-best move), but when testing our agent, we ran into some draws by threefold repetition that are avoided by that technique.

We further tried to avoid draws by forcing our agent to choose a different move in case one move leads so a draw. Of course, in the case of all validMoves leading to a draw, it is still possible for a game to end in a draw.

Our agent was told to avoid a move that leads to himself loosing in the next 4 moves (which was checked by the alpha-beta-pruning-algorithm).

We punished double pawns (2 pawns right above each other) by a certain value since double pawns take a big hit on mobility.

We tried to implement a good and fast way to let the mobility of all pieces also influence the evaluation of a move but found out that this would take too long with all pieces having calculated their own mobilityValue for each simulated move without making use of a structure like hashtables. However, we left the code for that in the file because that might be helpful for further

additions to the algorithm.

Note: Please keep the blank page here.